Malaysian Journal of Computer Science, Vol. 14 No. 1, June 2001, pp. 16-27

OPTIMAL DATAPATH DESIGN FOR A CRYPTOGRAPHIC PROCESSOR: THE BLOWFISH ALGORITHM

Noohul Basheer Zain Ali
Department of Electrical and Electronic Engineering
UTP, Bandar Seri Iskandar
31750 Tronoh, Perak
Malaysia

James M Noras
Department of Electrical and Electronic Engineering
Bradford University
Bradford BD7 1DP, UK
email: jmnoras@bradford.ac.uk

ABSTRACT

BLOWFISH is a fast cryptographic software algorithm, using the operations of addition, XOR and look-up tables. This paper reports on the design of a hardware implementation for greater speed, with pipelining and different bit-widths of registers and processing units. An 8-bit parallel data-path gives the best performance, mapping into 4 independent 8-bit modules, with a throughput at least 4 times greater than with 32-bit hardware. The design is suitable for implementation using small amounts of RAM and programmable logic.

Keywords: ***Digital design, Cryptography, Data encryption, Pipelining, Programmable logic, Feistel network, Adders***

1.0 INTRODUCTION

As we move into the twenty-first century, we see information processing and the telecommunication revolution continuing to grow very rapidly. The techniques of cryptography are essential for keeping information secret, for determining that information has not been tampered with and for controlling access to pieces of information [1]. Encryption is the transformation of data into a form unreadable by anyone without a secret decryption key [2], in order to ensure privacy. The information is hidden from anyone for whom it is not intended, even from those who can see the encrypted data. For example, one may encrypt files on a hard disk to prevent an intruder from reading them. Encryption also allows secure communication over an insecure channel. In a secure cryptosystem, the plaintext cannot be recovered from the ciphertext except by using the decryption key [2]. In a symmetric cryptosystem, a single key serves for encryption and decryption. The process is shown in Fig. 1.

1.1 The Blowfish Algorithm

Blowfish is a 64-bit block cipher presented by Bruce Schneier [3]: a software programmed in C, and is a suggested replacement for DES (Data Encryption Standard). DES was the standard cryptographic algorithm for more than 19 years, but it is now accepted that its key size is too small for present usage [4].

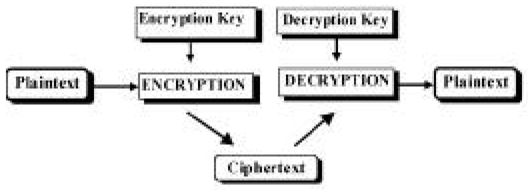


Fig. 1: Encryption and decryption

Blowfish is a fast algorithm and can encrypt data on 32-bit microprocessors at a rate of one byte every 26 clock cycles [5]. The algorithm is compact and can run in less than 5K of memory. It has a variable-length key block cipher of up to 448 bits. Although a complex initialisation phase is required, the encryption of data is very efficient on microprocessors. It suits applications where the key does not change often, for example, a communication link or automatic file encryptor.

1.3 The Potential of Hardware Implementation

There is a trend to produce software of designs of data encryption algorithms for the purpose of designer's understanding and debugging, but eventually to put them into hardware for maximum speed and increased security. Even though software implementation can be optimised using assembler code, the result is still slow compared with hardware implementations. For example, a software implementation of IDEA (International Data Encryption Algorithm) on a Sun SPARC2 workstation encrypts data at 400 Kbps, while a VLSI implementation of the same algorithm encrypts data at 177 Mbps, some 450 times faster [6].

1.4 Outline of Paper

The next section of this paper describes in detail the Blowfish algorithm. In its original form, it requires 32-bit processing. Section 3 describes how the algorithm is decomposed and reconstructed with smaller word sizes to find the best performance in terms of speed, and the implementation of the appropriate design using a pipeline method. Section 4 describes how the algorithm is mapped into 4 modules using scheduling and allocation techniques, tested and verified by simulation. The final section reviews the projected performance of the design and discusses potential applications and future work.

2.0 ANALYSIS OF THE BLOWFISH ALGORITHM

Blowfish is a symmetric block cipher that encrypts data in 8-byte (64-bit) blocks [3]. The algorithm has two parts, key expansion and data encryption. Key expansion consists of generating the initial contents of one array (the P-array), namely, eighteen 32-bit sub-keys, and four arrays (the S-boxes), each of size 256 by 32 bits, from a key of at most 448 bits (56 bytes). The data encryption uses a 16-round Feistel Network [7, 8]. This type of network dates from the early 70s [9, 10].

2.1 The Feistel Structure

Fig. 2 shows a block diagram of an algorithm with the Feistel structure for encryption, with 16 rounds of confusion and diffusion [11]. Mathematically the functions can be expressed as below:

> Take a block of 64 bits and divide it into two equal halves: L and R. Then define the iterated block cipher when the outputs L_i and R_i of the i'th round are determined from the outputs L_{i-1} and R_{i-1} of the previous round:
>
> $$L_i = L_{i-1}$$
> $$R_i = L_{i-1} \oplus F(R_{i-1}, K_i),$$
>
> where K_i is the sub-key used in the i'th round and F is specific to the particular algorithm.

The main feature of this construction is that it is reversible [12], in that the same network (with the same keys accessed in the reverse order) is used both for encryption and decryption. A Feistel-type network is not only used in Blowfish but it is also the basis of DES [13], and is used in many other cryptographic algorithms such as LOKI, GOST, FEAL, Lucifer, Khufu and Cast.

2.2 The F Function of Blowfish

The F Function, regarded as the primary source of algorithm security [3], combines two simple functions: addition modulo two (XOR) and addition modulo 2^{32}.

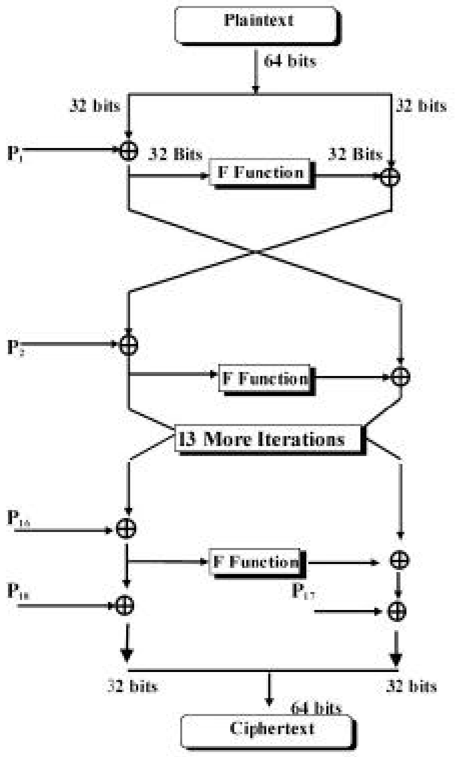


Fig. 2: Structure of Blowfish

Fig. 3 illustrates the F Function. Clearly, no complex mathematical process is involved, unlike exponentiation (modulo N) in RSA for example [14], but the mixture of the algebraically dissimilar types of operation makes inversion without knowledge of the key very difficult. The only additional operations are four indexed array data lookups per round.

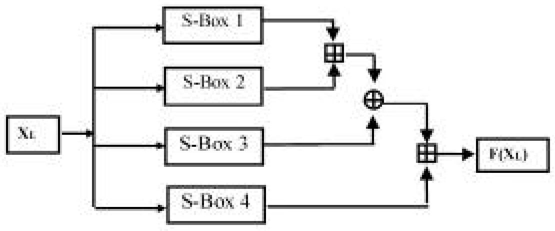


Fig. 3: F-function

The F function is the kernel and distinguishing feature of Blowfish [5], and is applied as follows:

Divide XL (32 Bits) into four 8-bit quarters: a, b, c, and d. Then:

F(XL)={(S1[a] + S2[b]) ⊕ S3[c]} + S[d])},

where + means addition modulo 2^{32}, and
⊕ means exclusive OR, i.e. XOR

S1[a], for example, means the content of S-box 1 at address a. The addresses a, b, c and d are 8-bits wide, while the S-box outputs are 32-bits wide. The process of data encryption can be described as in the pseudocode below:

```
Divide X into two 32-bit halves XL and XR
        For i=1 to 16:
               $XL = XL \oplus P_i$
               $XR = F(XL) \oplus XR$
               Swap XL and XR
        Endfor
Swap XL and XR      (Undo the last swap.)
        $XR = XR \oplus P_{17}$
        $XL = XL \oplus P_{18}$
Recombine XL and XR
Output X                    (64-bit data block: ciphertext)
```

For decryption, the same process is applied, except that the sub-keys P_i must be supplied in reverse order. The nature of the Feistel network ensures that every half is swapped for the next round (except, here, for the last two sub-keys P_{17} and P_{18}).

2.3 Speed and Area Factors

In each F function, there are two 32-bit additions. Since the F function loop is iterated 16 times for a block of data encryption, the additions are repeated 32 times. Not only does this arithmetic make the whole algorithm potentially very slow, but also 32-bit adders would require a large silicon area. Also, in the most obvious implementation, computation has to wait for the current process to finish before it can proceed to the next stage. For example, the first stage addition in the F function has to finish before the XOR function can be done. Similarly, the second addition cannot be executed until the previous XOR function is completed.

The iterations must remain at 16 rounds to ensure the security of the algorithm [5]. Thus, any speed-up requires a deeper analysis of the algorithm.

3.0 HARDWARE ANALYSIS

Pipelining is a well-known technique for improving the throughput of computers [15], by using parallel elements so that several instructions can be worked on simultaneously. The basic idea of pipelining is to begin carrying out a new instruction before execution of an old one is completed. When pipelining is used, the number of steps in the basic algorithm is less important than fitting the steps into a framework so that they can be performed in parallel [16].

Fig. 4 sets out one way of pipelining Blowfish, where the data are broken into two halves (High and Low).

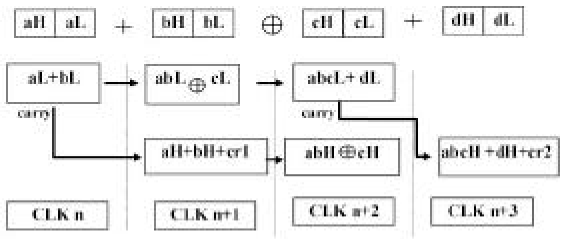


Fig. 4: Implementing pipelining with two parallel streams

These are processed separately (with delayed arithmetic carries – shown in the figure as carry) and then combined at the end of processing. Even though more clock cycles are needed, the speed of the clock can be greatly improved, because smaller adders are required at each stage, with smaller internal propagation delays. Note that the increase of speed requires an increase in silicon area, as registers are required to store intermediate results.

3.1 Scheduling and Allocation

The Blowfish system architecture has been analysed and designed using two interdependent procedures, scheduling and allocation. Scheduling is the process of assigning datapath operations to available time periods, and allocation is the association or binding of datapath operations to particular hardware resources. Different scheduling and allocation strategies can have significant effects upon the performance and hardware requirements of designs [17].

3.2 Different Word Size Processing

Processing for different word sizes has to be analysed to see the possibility of pipeline implementation of the design. Initially, the 32-bit level, which is used in the software version, is analysed.

3.2.1 32-bit Processing

The original software version uses 32-bit level processing. There are two different methods for implementation in 32-bit level processing: parallel and serial.

3.2.1.1 Parallel

The first method is by parallel implementation of the F function, so that the F function in each round together with the XOR in X_R, will require just one clock cycle, with no register to store intermediate results.

Fig. 5 shows the required configuration, which would require two 32-bit adders and two 32-bit XOR gates, and hence, a large area of silicon. Also, the clock would be slow since each 32-bit adder will have 32 levels of propagating carry, with propagation delays according to the type of adder used [18].

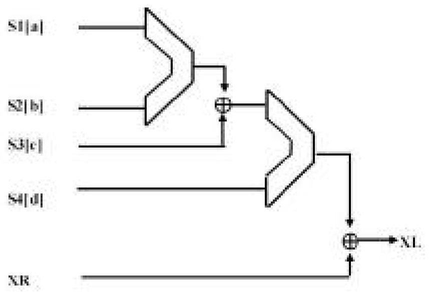
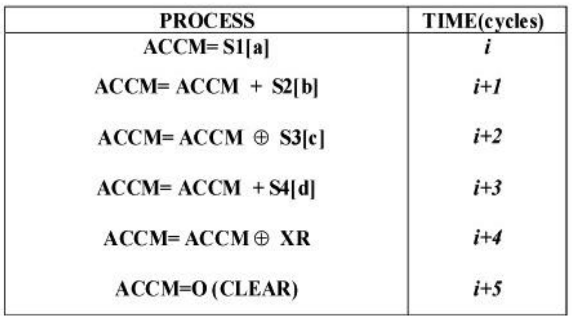


Fig. 5: Implementation of the F-function

A second approach is to compute the F function serially, i.e. every operation in a different clock cycle. In this approach, a 32-bit register is required to store the previous partial operation. This method requires six clock cycles for each round of F function. The critical timing diagram for this approach is shown in Fig. 6.



PROCESS	TIME(cycles)
ACCM= S1[a]	i
ACCM= ACCM + S2[b]	$i+1$
ACCM= ACCM ⊕ S3[c]	$i+2$
ACCM= ACCM + S4[d]	$i+3$
ACCM= ACCM ⊕ XR	$i+4$
ACCM=O (CLEAR)	$i+5$

Fig. 6: Critical timing for F function computed serially

A 32-bit adder/XOR is also required for this approach. The total time required to encrypt a block of data with a 16 round Feistel network will be 114 clock cycles:

6 (cycles for each F function)
times 16 (for each round of encryption) = 96

+18 (1 clock cycle for each XOR
with the P Arrays) = 18

= total of 114 clock cycles.

Table 1 shows the datapath for 32-bit processing. Blank boxes are unused (or wasted) states. However, since the longest process is 32-bit addition, the clock period has to be long enough to enable the propagation delay on the adder.

Table 1: Datapath for 32-bit processing

Clock	**P array XOR**	**Sbox Reading**	**Addition**	**XORing**	**Addition**	**XORing with xR**
1	P1 $\oplus$ xL1					
2		a1,b1,c1,d1				
3			a1+b1=ab1			
4				ab1+c1=abc1		
5					abc1+d1=Fout1	
6						Fout1 $\oplus$ xR1=xL2
7	P2 $\oplus$ xL2					
8		a2,b2,c2,d3				
9			a2+b2=ab2			
10				ab2+c2=abc2		
11					abc2+d2=Fout2	
12						Fout2 $\oplus$ xR2=xL3
13	P3 $\oplus$ xL3					
14		a3,b3,c3,d3				

In both the parallel and the serial approach, it is observed that the 32-bit addition influences the period of clock cycle and as a result slows down the speed of processing. Also, we have seen there is no possibility of implementing pipelining in the process. This is because the next round of the Feistel network cannot start before the last stage of the previous process finishes.

3.2.2 16-bit Processing

The 32-bit word processing could be broken into 16-bit level processing. In this level, the idea of pipelining can be implemented. All the 32-bit words have to be broken into 16-bit words, so this resembles the system shown in Fig. 4. This approach will take six clock cycles to process a 16-bit word as in the 32-bit level. The next 16-bit word will be processed in parallel, delayed by one clock cycle. However, before the end of the second 16-bit word process, the third 16-bit word processing can start. Therefore, the same number of clock cycles as in the 32-bit level will be required.

Since the addition is only 16-bit, the propagation delay caused by the propagating carry will be less than the 32-bit propagation delays. As a result, the clock speed can be faster and consequently the total processing time will be less, approximately half that of the 32-bit level. However, analysis of the data flow diagram shows that the proportion of unused states to used states is large. Moreover, 16-bit addition has still has a slow propagating carry. Thus, we continue to consider the 8-bit level.

3.2.3 8-bit Processing

At this level, all the processing words are 8-bit. By breaking down to byte level, there are more chances to fill the pipeline densely. This approach will also take six clock cycles for processing an 8-bit word. The next three 8-bit data will be processed in parallel, each delayed one clock cycle from the previous level. Table 2 shows the dataflow for 8-bit processing. Even though the first round takes a total of ten clock cycles, the next round will only take six clock cycles as in 32-bit and 16-bit designs.

Table 2: Datapath for 8-bit processing

Clock	**P array XOR**	**Sbox Reading**	**Addition**	**XORing**	**Addition**	**XORing with xR**
1	P11 $\oplus$ xL11	-	-	-	-	-
2	P12 $\oplus$ xL12	a11, a12, a13, a14		-	-	-
3	P13 $\oplus$ xL13	b11, b12, b13, b14	a11+0,a12+0, a13+0,a14+0	-	-	-
4	P14 $\oplus$ xL14	c11, c12, c13, c14	A11+b11 =ab11 &cr11"	-	-	-
5	-	d11, d12, d13, d14	a12+b12+cr11" =ab12 &cr12"	ab11 $\oplus$ c11 =abc11	-	-
6	-	-	a13+b13+cr12" =ab13 &cr13"	ab12 $\oplus$ c12 =abc12	abc11+d11 =abcd11+cr11'	-
7	-	-	a14+b14+cr14" =ab14	ab13 $\oplus$ c13 =abc13	abc12+d12+cr11' =abcd12	abcd11 $\oplus$ xR11 =xL21
8	P21 $\oplus$ xL21	-	-	ab14 $\oplus$ c14 =abc14	abc13+d13+cr12' =abcd13	abcd12 $\oplus$ xR12 =xL22
9	P22 $\oplus$ xL22	a21, a22, a23, a24	-	-	abc14+d14+cr13' =abcd14	abcd13 $\oplus$ xR13 =xL23
10	P23 $\oplus$ xL23	b21, b22, b23, b24	a21+0,a22+0, a23+0,a24+0	-	-	abcd14 $\oplus$ xR14 =xL24
11	P24 $\oplus$ xL24	c21, c22, c23, c24	A21+b21 =ab21 &cr21"	-	-	-
12	-	d21, d22, d23, d24	a22+b22+cr21" =ab22 &cr22"	ab21 $\oplus$ c21 =abc21	-	-
13	-	-	a23+b23+cr22" =ab23 &cr23"	ab22 $\oplus$ c22 =abc22	abc21+d21 =abcd21+cr21'	-
14	-	-	a24+b24+cr24" =ab24	ab23 $\oplus$ c23 =abc23	abc22+d22+cr21' =abcd22	abcd21 $\oplus$ xR21 =xL31

However, the processing time will be faster since all the logic will be 8-bit, including the additions, where the maximum carry will only propagate through eight stages of an adder. This will make the whole process at least 4 times faster than the 32-bit level, but 8-bit processing has additional advantages over 32-bit and 16-bit because routing will be less congested, making the hardware design faster and smaller.

This design will be analysed in detail and will be further modified to give its best performance.

3.3 Modification of the Feistel Network

In the original Feistel network, the output of the F function is XORed with XL before XORed with the next P array. For example, the output of the F function from stage one, Fout1, is XORed with XL then XORed with P2. This means the P array XORing cannot take place before the output of the F function appears. However, the processing time can be faster if the XORing with the P arrays is performed while the F function is still being evaluated. For this, we have to modify the Feistel network so that the XORing with the next P array is performed before the output of the F function appears. This is possible because XORing is commutative:

$$A \oplus B \oplus C = A \oplus C \oplus B$$

Another modification is to add a dummy function Fout01. The reason for this is to make the algorithm looks identical from the first round till the final round. This will make the decryption process, which takes the key in reverse order, much easier. An extra hardware requirement for the modified version is the addition of a few extra registers to keep the value of the previous stage's outputs for use in the next stage.

3.4 Combining S-boxes

From the datapath in Table 2, we can see that only one S-box is being read at each clock cycle. These S-boxes contain 256 locations each identified by different addresses, thus 1024 locations in total, each location being a 32-bit word. Therefore, these S-boxes can be combined into a RAM with 1024 locations. This will make the design easier as we do not have to control each RAM separately. The block diagram of the simplified system is shown in Fig. 7.

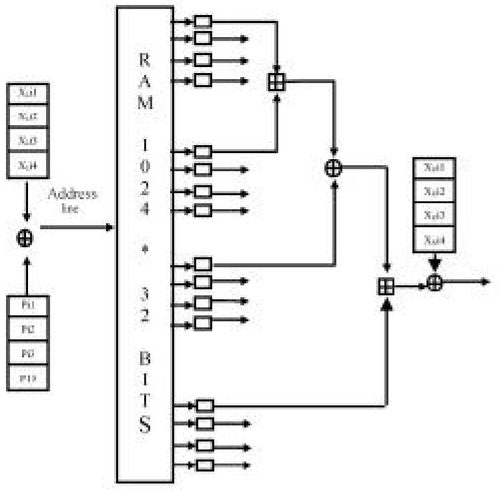


Fig. 7: 8-bit processing in modified structure

3.5 Datapath and Reducing the Number of Registers

After combining the S-boxes and modifying the Feistel Network, data flow analysis is repeated to see the new datapath. Table 3 shows the modified data flow. We can see that the efficiency of hardware is increased since the unused states decrease. Finally, we note that not all registers are used all the time, so that the number of the registers can be reduced at every stage except for the final one. For the first level, only one register is needed since only one byte of data, at maximum, is kept in any clock cycle. For the second level, two registers are needed, and three for the third level. The final level registers cannot be reduced since there are 4 registers used in maximum at certain clock cycles. The total number of the registers used is decreased from 16 registers to 10 registers.

4.0 IMPLEMENTATION

Best performance in terms of speed and hardware efficiency is given by 8-bit processing. Thus, the hardware configuration at this level was analysed before the integration for complete system.

An initial attempt to map to hardware suggests the following sequence of operations. Firstly, the P Array is XORed with the XR (right half of the data) and the result is XR'. Then XR' is XORed with the output of the accumulator (Fout) from the previous clock cycle. The result is then fed into RAM that will act as the RAM addressing values. A copy of these values is kept in the register to be used for subsequent rounds. The output of the RAM is kept in registers before it is fed into ADDER1 together with the carry from the previous module. The process continues with XORing, and lastly, addition using ADDER2. This cycle then continues for the successive rounds.

Table 3: Modified datapath for 8 -bit processing

Clock	P array XOR	XORing with xR	Sbox Reading	Addition	XORing	Addition
1	P11 $\oplus$ xL01 =xL11=xR21'	-	-	-	-	-
2	P12 $\oplus$ xL02 =xL12=xR22'	Fout01 $\oplus$ xR21' =xR21	-	-	-	-
3	P13 $\oplus$ xL03 =xL13=xR23'	Fout02 $\oplus$ xR22' =xR22	a11, a12, a13, a14	-	-	-
4	P14 $\oplus$ xL04 =xL14=xR24'	Fout03 $\oplus$ xR23' =xR23	b11, b12, b13, b14	a11+0,a12+0, a13+0,a14+0	-	-
5	-	Fouto4 $\oplus$ xR24' =xR24	c11, c12, c13, c14	a11+b11 =ab11 &cr11"	-	-
6	-	-	d11, d12, d13, d14	a12+b12+cr11" =ab12 &cr12"	ab11 $\oplus$ c11=abc11	-
7	P21 $\oplus$ xR11 =xR11'	-	-	a13+b13+cr12" =ab13 &cr13"	ab12 $\oplus$ c12=abc12	abc11+d11 =Fout11 +cr11'
8	P22 $\oplus$ xR12 =xR12'	Fout11 $\oplus$ xR11' =xL21=xR31	-	a14+b14+cr14" =ab14	ab13 $\oplus$ c13=abc13	abc12+d12+cr11' =Fout12+ cr12'
9	P23 $\oplus$ xR13 =xR13'	Fout12 $\oplus$ xR12' =xL22=xR32	a21, a22, a23, a24	-	ab14 $\oplus$ c14=abc14	abc13+d13+cr12' =Fout13+ cr13'
10	P24 $\oplus$ xR14' =xR14'	Fout13 $\oplus$ xR13' =xL23=xR33	b21, b22, b23, b24	a21+0,a22+0, a23+0,a24+0	-	abc14+d14+cr13' =Fout14
11	-	Fout14 $\oplus$ xR14' =xL24=xR34	c21, c22, c23, c24	a21+b21 =ab21 &cr21"	-	-
12	-	-	d21, d22, d23, d24	a22+b22+cr21" =ab22 &cr22"	ab21 $\oplus$ c21=abc21	-
13	P31 $\oplus$ xR21 =xR21'	-	-	a23+b23+cr22" =ab23 &cr23"	ab22 $\oplus$ c22=abc22	abc21+d21 =Fout21+cr21'
14	P32 $\oplus$ xR22 =xR22'	Fout21 $\oplus$ xR21' =xL31=xR41	-	a24+b24+cr23" =ab24	ab23 $\oplus$ c23=abc23	abc22+d22+cr21' =Fout22+ cr22'

However, the design can be simplified by using the same hardware block for multiple functions. The following functions are performed in different clock cycles:

1. addition in ADDER1,
2. XOR with output from ADDER1 and Register,
3. addition in ADDER2, and
4. XOR between first XOR and Fout.

All of these operations can be done by using just one hardware block that combines the XOR and ADD functions. We introduce a device called ADD/XOR, which is essentially a custom ALU. By doing this, not only will the system look simpler but the area of silicon for fabricating the system will also decrease.

4.1 The ADD/XOR Block

Since the adder is an important factor in influencing the speed of the algorithm, it is very important to analyse the types of adders available. Generally, the type of adder is determined by the way it handles the carry. The main types of adders are Ripple Carry Adder and Look Ahead Carry Adder.

4.1.1 Ripple Carry Adder

Ripple carry adders are adders where the carry output of each full-adder [19] is connected to the carry input of the next higher-order stage. The sum and carry of any successive stage are not stable until all previous carries have occurred, which lead to time delay in the addition process. For an adder of length more than a very few bits, this delay is likely to be unacceptable.

4.1.2 Carry-lookahead

One method of eliminating this ripple carry delay is called carry-lookahead. This method is based on two functions of the full-adder, called the carry-generate and carry-propagate functions [19]. Carry-generate is expressed as an AND function and carry-propagate is expressed as an OR function. Using this method, the carry does not propagate through all the adders but passes through the AND and OR gates, which are built using the concept of carry propagate and carry generate. The trade-off for this method is increased area and complexity. Thus, the method is not extendable indefinitely to very long adders, as the complexity, and hence, propagation of the additional logic increases. In the present case, it would be unfeasible to build 16-bit or 32-bit adders, although the method is ideal for an 8-bit system.

4.1.3 Combining XOR and Adder Functions

The adder circuit can be modified into the ADD/XOR device in the system. Consider a 2-stage adder [19]. The Boolean equation for digit 1 of the addition is:

i'th stage $\Sigma_i = A_i \oplus B_i \oplus C_{in\_1}$

If the incoming carry is ignored, then the result will be only the XOR function between numbers A and B, so the modified adding equation needed to implement the F function would be:

i'th stage $\Sigma_i = A_i \oplus B_i \oplus (C_{in\_1} \,\&\, \text{mode})$

Here, & means Boolean AND. Clearly, if mode = 0 for all the stages, then, $\Sigma = A \oplus B$. In this way, the adder circuit can be modified into an ADD/XOR device. Since we are using the same device for different functions we need to introduce a register that will act as an accumulator to hold data from the previous stage before feeding it to the next stage. Fig. 8 shows the block diagram of this combined structure.

If mode = 1:	ADD/XOR functions as an adder
If mode = 0:	ADD/XOR functions as a bank of XOR gates.

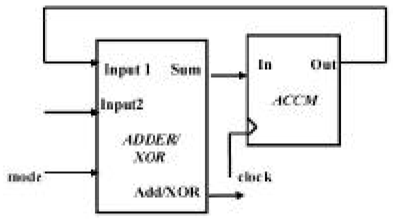


Fig. 8: Combined adder and accumulator

Fig. 9 shows the modified hardware configuration for 8-bit processing with the ADD/XOR included.

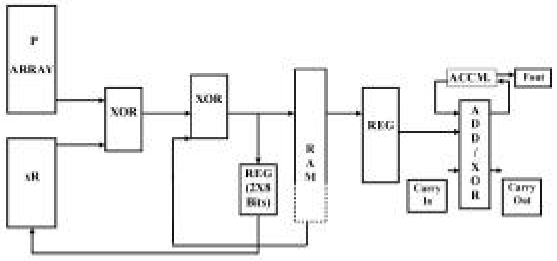


Fig. 9: Hardware configuration for 8-bit processing

4.2 Mapping the System

The complete system is mapped into 4 independent modules that are linked by the RAM and the carries, with the P array split into different sections. This is possible since the P array has its own addressing and can be divided for each module. No modification is required for other parts of the design since all other processing is 8-bit.

4.3 Throughput Estimation

Exact throughput can be obtained only when the complete system is mapped into a particular technology. However, it is useful to obtain estimates, using the speed of clock cycle as presented by Sommerville [20]. The 32-bit level needs 144 clock cycles to encrypt 64 bits of data [see section 3.2.1.1]. For the 8-bit level, the number of clock cycles is also 144. However, since the processing will be 8-bits rather than 32-bits, latency, which is the measure of processing rate [21], will be at least 4 times less than the 32-bit version. This is because there is less propagating carry in the adders (even when using carry-lookahead). Also, there will be less routing complexity, whether using VLSI or programmable logic.

5.0 CONCLUSION

Although Blowfish is a fast encryption algorithm, hardware implementation can make the system throughput much higher. After validating the design by extensive simulation, we made provisional mapping of the to Xilinx programmable logic FPGAs, to show that the complexity of design can be accommodated with this technology, using off-chip RAM for highest performance. This is an important practical consideration, since it allows security hardware to be built into digital systems without the expense of VLSI design and fabrication, and to permit design upgrades at low cost.

The paper proves the feasibility of a pipelined hardware implementation of Blowfish. Pipelining can be implemented in 16-bit and 8-bit sizes with the latter offering greater linearity. An 8-bit system appears promising because of speed enhancement with little expansion of silicon area. The predicted throughput is at least 4 times greater than the original 32-bit version.

REFERENCES

[1] S. A. Venstone, A. Menezes, *Handbook of Cryptography*. CRC Press, 1996.

[2] Bruce Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* 2nd Edition, New York, John Wiley and Sons, Inc. 1996, pp. 21-27.

[3] Bruce Schneier, “Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)”, in *Cambridge Security Workshop on Fast Software Encryption*, Cambridge, UK, December 9-11, 1993, pp. 191-204.

[4] Dr. Dobbs, “Blowfish - One Year Later”, *<http://www.counterpane.com/bfdobsoyl.html>*, 1996.

[5] Bruce Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* 2nd Edition, New York, John Wiley and Sons, Inc. 1996, p. 339.

[6] R. Zimmerman, A. Curiger, H. Bonnenberg, H. Kaeslin, N. Felber and W. A. Fitcher, “177 Mb/s VLSI Implementation of the International Data Encryption Algorithm”. *IEEE Journal of Solid State Circuits*, Vol. 29, No. 3, 1994, pp. 303-307.

[7] H. Feistel, and W. A. Notz, “Some Cryptographic Techniques for Machine to Machine Data Communication”, in *Proceedings of the IEEE*, Vol. 63, No. 11, 1975, pp. 1545-1554.

[8] H. Feistel, “Cryptography and Computer Privacy”. *Scientific American*, Vol. 228, No. 5, 1973, pp. 15-23.

[9] H. Feistel, “Block Cipher Cryptographic System”. *U. S. Patent #3798605*, 19 March 1974.

[10] H. Feistel, "Cryptography and Computer Privacy". *Scientific American*, Vol. 228, No. 5, 1973, pp. 15-23.

[11] W. Peter, *Disappearing Cryptography*. Massachusetts, AP Professional, 1996.

[12] Bruce Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. 2nd Edition, New York, John Wiley and Sons, Inc. 1996, p. 347.

[13] B. Henry and P. Fred, *Cipher Systems: The Protection of Communication*. London, Northwood Books, 1982.

[14] Bruce Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. 2nd Edition, New York, John Wiley and Sons, Inc. 1996, pp. 466-468.

[15] H. Taub, *Digital Circuits and Microprocessors*. New York, McGraw-Hill Book Company, 1992, pp. 403-405.

[16] T. C. Bartee, *Computer Architecture and Logic Design*. Singapore, McGraw-Hill, 1991, pp. 476-479.

[17] P. G. Paulin and J. P. Knight, "Algorithms for High-Level Synthesis". *IEEE Design and Test of Computers*, Dec. 1989, pp. 15-28.

[18] D. Lewin, *Theory and Design of Digital Computer Systems*. Surrey, Eng. Thomas Nelson and Sons Ltd. 1980, pp. 131-152.

[19] T. L. Floyd, *Digital Fundamentals*. 4th Edition, New York, Macmillan Publishing Company, 1990, pp. 239-243.

[20] I. Sommerville, *Software Engineering*. 3rd Edition, London, Addison-Wesley Publishing Company, 1989, p. 308.

[21] Bruce Schneier, "Block Cipher Speed Comparison", *<http://www.counterpane.com/speed.html>*, Feb. 1997, Accessed 5 April 1997.

BIOGRAPHY

Noohul Basheer Zain Ali gained a Diploma in Electronic Engineering from Universiti Teknologi Malaysia in 1995, and graduated with BEng (Honours) in Electronic, Communication and Computer Engineering in 1997 from the University of Bradford, UK. He is currently studying at the Rensselaer Polytechnic Institute, New York, US, for a MSc in Computer Systems. From 1998-2000 he worked for Petronas, and is currently a Lecturer in the Department of Electrical and Electronic Engineering, Universiti Teknologi Petronas, Bandar Seri Iskandar, 31750, Tronoh, Perak, Malaysia. His current research interests are logic design, DSP and VLSI implementation for Artificial Intelligence.

James McKenzie Noras graduated with BSc (Physics) from St Andrews University (1973), PhD (Semiconductor Physics) from St Andrews University (1978) and MSc (Mathematics) from Open University (1995). From 1987, he has been a lecturer in the Department of Electronic and Electrical Engineering, Bradford University, UK, where he is presently Chairman of the Undergraduate Courses Academic Board. He is a member of the VCE in Personal and Mobile Communications, with research interests in DSP for reconfigurable mobile communications.